

# RVS (RISC-V Visual Simulator)

## Assembler Manual v0.10

### 1. Syntax

Every instruction or assembly command must be on a separate line and cannot be continued on a new line.

Every line starts with an optional label, followed by an instruction/command mnemonic code, followed by a comma-separated list of parameters, followed by an optional comment.

Labels, when defined, are sequences of alphanumeric characters that start with a letter and are terminated by a colon (the colon is not a part of the label).

The field separators in the list of parameters, depending on the instruction, are commas, space characters, and open and close parentheses.

Comments are preceded by a semi-colon character ( ; ), a number sign ( # ), or by two slash characters ( // ). The comment always continues to the end of the line. If it starts at the beginning of the line then the entire line is a comment line.

## 2. Arrangement of code and data in memory

The assembly begins at the default memory address of 0. This can be changed by the `ORG` assembly command.

Each assembled machine instruction takes one word (4 bytes) so the memory pointer is increased by 4. Machine instructions are automatically aligned at word boundaries (the address is divisible by 4.)

Each assembled **Define** directive (`DD`, `DW`, `DH`, `DB`, `DC`, `DM`) takes one or more double-words so the memory pointer is increased by a multiple of 8 bytes. **Define** directives are automatically aligned at double-word boundaries (the address is divisible by 8). Unused bits are filled with 0s.

There are no restrictions on how the code and data are arranged in the memory as long as there is no overlapping. In case of overlapping RVS will issue an error message and stop the source text processing right at the overlapping line.

The default starting address of the assembled program, which is the initial value of the program counter (PC), is the instruction labeled `START`. If there is no instruction labeled `START`, then the initial value of the PC is the address of the first compiled machine instruction. The first compiled machine instruction is the first machine instruction seen by the compiler as it goes through the assembly file.

### 3. Constants

The following formats can be used for specifying constants.

- **Binary:** `0b010`, `0B10`  
Starts with `0b` or `0B` and contains only the characters `{01}`. An optional sign can be inserted before the leading `0b` or `0B`.
- **Octal:** `010`, `017` (10 and 17 are decimal, not octal)  
Starts with `0` and contains only the digits `{01234567}`. An optional sign can be inserted before the leading `0`.
- **Decimal:** `123`, `4567` (0123 and 04567 are octal, not decimal)  
Starts with a non-zero digit and contains only the digits `{0123456789}`. An optional sign can be inserted before the leading digit.
- **Hexadecimal:** `0x010`, `0X10`, `0x1a`, `0xCD`  
Starts with `0x` or `0X` and contains only the following characters `{0123456789abcdefABCDEF}`. An optional sign can be inserted before the leading `0x` or `0X`.
- **Characters:** `"101"`, `"0101"`, `"123"`, `"abcd"`, `"Test, test\n\0"`  
Constitutes a sequence of characters enclosed in double quotes. Each character is converted to its 8-bit ASCII code. Backslash substitutions are also supported e.g. `"\n"` becomes Newline (`0xA`). Character codes can be specified in octal e.g. using `"\0"` for NULL terminated strings. The octal specification `"\12"` generates the same value (`0xA`) as the Newline `"\n"` specification. Note that the terminating null character is not inserted automatically; if you want to include it you must specify it in the string.

## 4. Integers

If an integer number is given without a sign, or with a positive sign, it is treated as an unsigned integer (meaning the most significant bit could be 1 if the number is big enough.) For example if we try to store 65000 in a half-word, it is directly converted to the hexadecimal value of 0xFDE8 (the MSB is 1, but the number has been treated as unsigned).

If an integer number is given a negative sign, it is stored in 2's complement format. For example, if we try to store -65000 in a half-word the 2's complement of it, which is 0xFFFFFFFF0218 (in 64-bit format) is calculated. The obtained result is then truncated to 16 bits and the value of 0x0218 is stored (that is the decimal value of 536).

If a given integer number (positive or negative) does not fit in the space allocated, the excessive most significant bits will be truncated. For example if we want to store 165000 in a half-word, it clearly will not fit (the largest 16-bit unsigned integer number is 65535), then since 165000 in hexadecimal is 0x28488, the leading 2 will be truncated and the value of 0x8488 will be stored (that is the decimal value of 33928).

## 5. Define directives

The define directives reserve memory space and initialize it with given values. We can define data of different lengths, e.g. double-word (64 bits), word (32 bits), half-word (16 bits), and bytes and characters (8 bits).

**DD (Define Double-words):** It reserves memory and initializes 64-bit double-words. The value could be given in binary (e.g. 0b10101), octal (octal numbers start with 0), decimal (decimal numbers start with a non-zero digit), and hexadecimal (e.g. 0x12aa). DD accepts multiple constants separated by commas that are stored in consecutive 64-bit double-words (1 constant per double-word.)

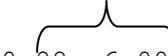
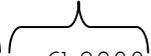
```
DD    0b010, 010, 10, 17, 0x010, -0b10, -010, -10, -0x10
```

Memory address	Content		Decimal
Double word 0 0x0000000000000000	0x0000000000000002	0b010	2
Double word 8 0x0000000000000008	0x0000000000000008	010	8
Double word 16 0x0000000000000010	0x000000000000000a	10	10
Double word 24 0x0000000000000018	0x0000000000000011	17	17
Double word 32 0x0000000000000020	0x0000000000000010	0x010	16
Double word 40 0x0000000000000028	0xfffffffffffffffe	-0b10	-2
Double word 48 0x0000000000000030	0xffffffffffffff8	-010	-8
Double word 56 0x0000000000000038	0xffffffffffffff6	-10	-10
Double word 64 0x0000000000000040	0xffffffffffffff0	-0x10	-16

The values are stored at memory addresses in increasing order. The sixth constant (-0b10) in the above DD directive is stored, for example, at address 0x28 (address 40 in decimal.) As shown in row 6 of the above table the stored value is the 2's complement of 0b10 (2 in decimal) which is 0xfffffffffffffffe (-2 in decimal).

**DW (Define Words):** It reserves memory and initializes 32-bit words. The value could be given in binary, octal, decimal, or hexadecimal. It accepts multiple constants that are stored in consecutive 32-bit words (2 constant per double-word in little endian order.) Every DW directive starts at double-word boundary (multiple of 8) and, if necessary, the last double-word is padded with 0s.

**DW 0b010, 17, 0x20, 2000000000, 4000000000, -2000000000**

Memory Address	Content	
Double word 0 0x0000000000000000	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">           Word 4              0x00000011         </div> <div style="text-align: center;">           Word 0              00000002         </div> </div>	Word 0 contains 0b010. Word 4 contains 17.
Double word 8 0x0000000000000008	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">           Word 12              0x77359400         </div> <div style="text-align: center;">           Word 8              00000020         </div> </div>	Word 8 contains 0x20. Word 12 contains 2000000000 (0x77359400) as unsigned number.
Double word 16 0x0000000000000010	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">           Word 20              0x88ca6c00         </div> <div style="text-align: center;">           Word 16              ee6b2800         </div> </div>	Word 16 contains 4000000000 (0xee6b2800) as unsigned number. Word 20 contains the 2's complement of 2000000000 (0x77359400) which is (0x88ca6c00) in hexadecimal.

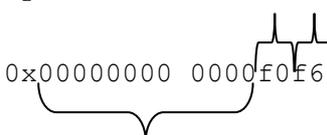
**DH (Define Half-words):** It reserves memory and initializes 16-bit half-words. The value could be given in binary, octal, decimal, or hexadecimal. It accepts multiple constants that are stored in consecutive 16-bit half-words (4 constant per double-word in little endian order.) Every DH directive starts at double-word boundary (multiple of 8) and, if necessary, the last double-word is padded with 0s.

**DH 0b010, 010, 017, 10, 17, 0x010,-0b10,-010,-10,-0x10**

Memory Address	Content	
Double word 0 0x0000000000000000	<p>HW 6 4 2 0 0x000a000f 00080002</p>	HW0=0b10 (=0x0002) HW2=010 (octal 8=0x0008) HW4=017 (=8+7=15=0x000f) HW6=10 (=0x000a)
Double word 8 0x0000000000000008	<p>HW 14 12 10 8 0xfff8ffffe 00100011</p>	HW8=17 (=0x0011) HW10=0x0010 HW1=-0b10 (-2=0xffffe) HW14=-010 (=-8=0xffff8)
Double word 16 0x0000000000000010	<p>HW 22 20 18 16 0x00000000, fff0ffff6 Padded with 0s</p>	HW16=-10 (=0xffff6) HW18=-0x10 (=-16=0xff00) HW20 and HW22 are padded with 0s.

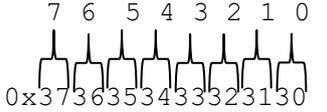
**DB (Define Byte):** It reserves memory and initializes 8-bit bytes. The value could be given in binary, octal, decimal, or hexadecimal. It accepts multiple constants that are stored in consecutive 8-bit bytes (8 constant per double-word in little endian order.) Every DB directive starts at double-word boundary (multiple of 8) and, if necessary, the last double-word is padded with 0s

**DB     0b010, 010, 017, 10, 17, 0x010, -0b10, -010, -10, -0x10**

Memory Address	Content	
Double word 0 0x0000000000000000	Byte 6 5 4 3 2 1 0  0xf8fe1011 0a0f0802	Byte 0=2. Byte 1=8. Byte 2=15. Byte 3=10. Byte 4=17. Byte 5=16. Byte 6=-2. Byte 7=-8.
Double word 8 0x0000000000000008	Byte 15..10 9 8  0x00000000 0000f0f6 Padded with 0s	Byte 8=-10. Byte 9=-16. Bytes 15..10 are padded with 0s.

**DC (Define Characters):** Accepts a single constant which must be a sequence of characters enclosed in double quotes. The ASCII codes of the characters are stored in one or more consecutive 64-bit double-words (8 characters per double-word) in increasing order of memory addresses. If necessary, the last double-word is padded with 0s. Include a trailing "\0" in the sequence of characters to make sure that it will be null-terminated irrespectively of its length. Note that the assembler will not automatically insert the terminating null character; you must specify it yourself.

**DC     "0123456789\0"**

Memory Address	Content	
Double word 0 0x0000000000000000	7 6 5 4 3 2 1 0  0x3736353433323130	Byte 0="0" (=0x30). Byte 1="1" (=0x31). ...
Double word 8 0x0000000000000008	Padded 0s \0 9 8  0x0000000000003938	Byte 8="8" (=0x38). Byte 9="9" (=0x39). Byte 10="\0" (=0x0).

**DM (Define Memory in double-words):** Initializes the specified number of 64-bit double-words with all 0s.

**DM 3**

<b>Memory Address</b>	<b>Content</b>
Double word 0 0x0000000000000000	0x0000000000000000
Double word 8 0x0000000000000008	0x0000000000000000
Double word 16 0x0000000000000010	0x0000000000000000

## 6. The ORG (Origin) directive

**ORG (Origin):** Defines the address at which the next assembled data or instruction will be placed.

```
ORG 0x1000
DD 1
ORG 0x800
addi x5,x0,1
```

The `ORG 0x1000` changes the current code/data generation address to `0x1000`, so the next data/instruction is placed in that address. Since the next data is `DD 1`, the constant `1` is placed in the double word at the address of `0x1000`.

The second `ORG 0x800` changes the current code/data generation address to `0x800`, so the next instruction (`addi`) is placed at the address of `0x800`.

An attempt to place any code or data in a memory location that has already been used triggers a compilation error.

The memory layout of the previous code segment is

### ASSEMBLY LISTING

ADDRESS	BIN/HEX CODE	TEXT SOURCE
0x0000000000000800	I 000000000001 0000 000 00101 0010011	addi x5,x0,1
0x0000000000001000	DD 0x0000000000000001	DD 1

### SYMBOL TABLE

0x0000000000000800 START

Since the first compiled executable statement is at location `0x800`, the `START` label is assigned the value of `0x800`.

## List of supported RISC-V instructions

	31	27	26	25	24	20	19	15	14	12	11	7	6	0	
	funct7		rs2			rs1		funct3		rd		opcode			R-type
	imm[11:0]					rs1		funct3		rd		opcode			I-type
	imm[11:5]		rs2			rs1		funct3		imm[4:0]		opcode			S-type
	imm[12 10:5]		rs2			rs1		funct3		imm[4:1 11]		opcode			B-type
	imm[31:12]					rd		opcode			U-type				
	imm[20 10:1 11 19:12]					rd		opcode			J-type				

### RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI	
imm[31:12]					rd	0010111	AUIPC	
imm[20 10:1 11 19:12]					rd	1101111	JAL	
imm[11:0]		rs1			000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011			BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011			BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011			BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011			BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011			BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011			BGEU
imm[11:0]		rs1			000	rd	0000011	LB
imm[11:0]		rs1			001	rd	0000011	LH
imm[11:0]		rs1			010	rd	0000011	LW
imm[11:0]		rs1			100	rd	0000011	LBU
imm[11:0]		rs1			100	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011			SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011			SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011			SW
imm[11:0]		rs1			000	rd	0010011	ADDI
imm[11:0]		rs1			010	rd	0010011	SLTI
imm[11:0]		rs1			011	rd	0010011	SLTIU
imm[11:0]		rs1			100	rd	0010011	XORI
imm[11:0]		rs1			110	rd	0010011	ORI
imm[11:0]		rs1			111	rd	0010011	ANDI
0000000		rs2	rs1	000	rd	0110011	ADD	
0100000		rs2	rs1	000	rd	0110011	SUB	
0000000		rs2	rs1	001	rd	0110011	SLL	
0000000		rs2	rs1	010	rd	0110011	SLT	
0000000		rs2	rs1	011	rd	0110011	SLTU	
0000000		rs2	rs1	100	rd	0110011	XOR	
0000000		rs2	rs1	101	rd	0110011	SRL	
0100000		rs2	rs1	101	rd	0110011	SRA	
0000000		rs2	rs1	110	rd	0110011	OR	
0000000		rs2	rs1	111	rd	0110011	AND	
0000000000000			00000	000	00000	1110011	ECALL	
0000000000001			00000	000	00000	1110011	EBREAK	

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7			rs2		rs1		funct3		rd		opcode			<b>R-type</b>
imm[11:0]					rs1		funct3		rd		opcode			<b>I-type</b>
imm[11:5]			rs2		rs1		funct3		imm[4:0]		opcode			<b>S-type</b>

**RV64I Base Instruction Set (in addition to RV32I)**

imm[11:0]			rs1		110		rd		0000011			<b>LWU</b>	
imm[11:0]			rs1		011		rd		0000011			<b>LD</b>	
imm[11:5]			rs2		rs1		011		imm[4:0]		0100011		<b>SD</b>
000000		shamt		rs1		001		rd		0010011			<b>LLI</b>
000000		shamt		rs1		101		rd		0010011			<b>SRLI</b>
010000		shamt		rs1		101		rd		0010011			<b>SRAI</b>

**RV32M Standard Extension**

0000001		rs2		rs1		000		rd		0110011			<b>MUL</b>
0000001		rs2		rs1		001		rd		0110011			<b>MULH</b>
0000001		rs2		rs1		010		rd		0110011			<b>MULHSU</b>
0000001		rs2		rs1		011		rd		0110011			<b>MULHU</b>
0000001		rs2		rs1		100		rd		0110011			<b>DIV</b>
0000001		rs2		rs1		101		rd		0110011			<b>DIVU</b>
0000001		rs2		rs1		110		rd		0110011			<b>REM</b>
0000001		rs2		rs1		111		rd		0110011			<b>REMU</b>